附录 B　标准库概要

“如果可能，所有复杂性都应埋藏于视野之外。”

——David J. Wheeler

本附录概述重要的 C++ 标准库特性。本附录内容都是精心选择的，特别适合那些希望接触一些本书之外内容的初学者。

B.1　概述

本附录的目的是作为补充参考资料，而不是像其他章节一样需要从头到尾仔细阅读。它（或多或少）系统地描述了 C++ 标准库的一些重要特性。本附录不是完整的参考资料，而只是一些重要特性的概述。通常，你需要查看相关章节来获得更为详细、完整的解释。注意，本附录不追求与 C++ 标准相同的精确性和术语，而是追求易于查阅。更详细的信息可参考 Stroustrup 的《The C++ Programming Language》一书。ISO C++ 标准中有标准库的完整定义，但它并不是为了初学者所编写的，因此不适合入门阅读学习。不要忘了使用联机帮助来查找标准库的有关内容。

一个选择性的（因而不完整的）概要介绍有什么用处呢？你可以从中快速查找已经知道的特性，也可以快速浏览一节来了解标准库中有哪些常用的特性。你可能必须到其他地方查找细节内容，但这没有关系：通过本附录，你已经获得了“查找什么”的线索。而且，本附录包含了交叉引用，你可以迅速找到包含详细内容的章节。本附录是 C++ 标准库特性的一个简洁概述。请不要尝试记忆本附录中的内容，这不是本附录的目的，相反，本附录是一个工具，能帮助你避免形成错误的记忆。

你可以在本附录中找到所需要的有用的特性，不要试图自己重新发明。标准库中的所有特性（特别是本附录中所提到的特性），已经被证明对很多程序员来说都是有用的工具。标准库中的工具，几乎总是比你仓促设计和实现得到的工具有着更为良好的设计、实现、文档以及更好的可移植性。因此，只要可能，你应该优先使用标准库特性，而不是“自制工具”。这样做，你的代码就更容易被他人所理解。

如果你是个理智的人，你会觉得本附录介绍的内容太多了，不要担心，忽略那些你不需要的内容就是了。如果你是个“细节狂人”，你会觉得本附录缺少了很多内容，但是，完整性是那些专家级教材和指南的目标，而且联机帮助中已经有足够完整的信息了。无论你是哪种人，你都会发现很多看起来很神秘，而且可能很有趣的内容。试着认真研究其中一些内容！

B.1.1　头文件

标准库的接口都是在头文件中定义的。下表列出的头文件中，有一些不在 C++ 1998 ISO 标准中，但它们都已被广泛使用，会被包含在下一版标准（C++0x）中。使用这些头文件中定义的功能可能需要安装/使用不同于 std 的名字空间（如 tr1 或 boost）。请将本节作为 C++ 标准库的一个概览，它可以帮助你找到某个功能的定义和描述。

STL(容器、迭代器和算法)	
< algorithm >	算法；sort( )、find( )等(参见附录 B. 5 和 21. 1 节)
< array >	固定大小的数组(C++0x)(参见 20. 9 节)
< bitset >	bool 数组(参见 25. 5. 2 节)
< deque >	双端队列
< functional >	函数对象(参见附录 B. 6. 2)
< iterator >	迭代器(参见附录 B. 4. 4)
< list >	双向链表(参见附录 B. 4 和 20. 4 节)
< map >	(key, value)的映射和多重映射(参见附录 B. 4 和 21. 6. 1 ~21. 6. 3 节)
< memory >	供容器值用的分配器
< queue >	队列和优先队列
< set >	集合和多重集合(参见附录 B. 4 和 21. 6. 5 节)
< stack >	栈
< unordered_map >	散列映射(C++0x)(参见 21. 6. 4 节)
< unordered_set >	散列集合(C++0x)
< utility >	运算符和 pair(参见附录 B. 6. 3)
< vector >	向量(可动态扩展)(参见附录 B. 4 和 20. 8 节)

I/O 流	
< iostream >	I/O 流对象(参见附录 B. 7)
< fstream >	文件流(参见附录 B. 7. 1)
< sstream >	string 流(参见附录 B. 7. 1)
< iosfwd >	声明(但未定义)I/O 流工具
< ios >	I/O 流基类
< streambuf >	流缓冲
< istream >	输入流(参见附录 B. 7)
< ostream >	输出流(参见附录 B. 7)
< iomanip >	格式化和操纵符(参见附录 B. 7. 6)

字符串操作	
< string >	string(参见附录 B. 8. 2)
< regex >	正则表达式(C++0x)(参见第 23 章)

数值计算	
< complex >	复数及其算术运算(参见附录 B. 9. 3)
< random >	随机数发生器(C++0x)
< valarray >	数值数组
< numeric >	通用数值算法，如 accumulate( )(参见附录 B. 9. 5)
< limits >	数值限制(参见附录 B. 9. 1)

工具和语言支持	
< exception >	异常类型(参见附录 B. 2. 1)
< stdexcept >	异常层次(参见附录 B. 2. 1)
< locale >	本地化格式
< typeinfo >	标准类型信息(从 typeid 获取)
< new >	内存分配和释放函数

C 标准库	
< cstring >	C 风格字符串操作(参见附录 B. 10. 3)
< cstdio >	C 风格 I/O(参见附录 B. 10. 2)
< ctime >	clock( )、time( )等(参见附录 B. 10. 5)
< cmath >	标准浮点数学函数(参见附录 B. 9. 2)
< cstdlib >	其他函数：abort( )、abs( )、malloc( )、qsort( )等(参见第 27 章)
< cerrno >	C 风格错误处理(参见 24. 8 节)
< cassert >	断言宏(参见 27. 9 节)

（续）

C 标准库	
< clocale >	本地化格式
< climits >	C 风格数值限制（参见附录 B. 9. 1）
< cfloat >	C 风格浮点限制（参见附录 B. 9. 1）
< cstddef >	C 语言支持：size_t 等
< cstdarg >	可变参数宏
< csetjmp >	setjmp( ) 和 longjmp( )（不要使用）
< csignal >	信号处理
< cwchar >	C 的宽字符
< cctype >	字符分类（参见附录 B. 8. 1）
< cwctype >	宽字符分类

每个 C 标准库头文件都有一个不带开始字母 c 并有 . h 后缀的版本，例如 < ctime > 的另一个版本是 < time. h > 。. h 版本定义的名字都位于全局空间，而不是 std 名字空间。

在后面几节中（以及前面一些章节中），我们将介绍这些头文件中的一些（但不是所有的）特性。如果你希望了解更多内容，请查阅联机帮助或者专家级 C++ 书籍。

B. 1. 2　名字空间 std

标准库特性都定义于名字空间 std 中，因此使用时需显式使用限定符，如使用 using 声明或 using 指令：

```
std::string s;              // explicit qualification

using std::vector;          // using declaration
vector<int>v(7);

using namespace std;        // using directive
map<string,double> m;
```

在本书中，我们采用 using 指令。使用 using 指令要有节制，参见附录 A. 15。

B. 1. 3　描述风格

如果想完整描述一个标准库操作，即使是一个简单操作，如一个构造函数或一个算法，都要花费几页的篇幅。因此，我们采用一种非常简化的描述风格。例如：

描述方法示例	
p = op(b, e, x)	对区间[b:e)和 x 进行 op 操作，将返回值赋予 p
foo(x)	对 x 进行 foo 操作，不返回结果
bar(b, e, x)	对 x 和区间[b:e)进行一些操作，结果为真

我们尽量使用一些含义明确的助记符（如 b、e 表示迭代器）指出一个范围；p 表示指针或迭代器；x 表示值；当然，所有这些含义都与上下文相关。在这种表示方法中，只有借助注释才能分辨不返回值和返回布尔值这两种情况，因此有可能产生混淆。对于返回布尔值的操作，注释通常以问号结束。

如果算法遵循习惯方式，通过返回输入序列结束标记来表示“故障”、“未找到”等含义（参见

B. 3. 1)，我们不再重复说明。

B. 2　错误处理

标准库中不同部分的开发时间可能相差 40 年，因此错误处理的风格和方法是不一致的。

- C 风格库由函数构成，这些函数中很多都是通过设置 errno 来表示发生了错误，参见 24. 8 节。
- 很多对元素序列进行操作的算法会返回一个指向末尾元素之后位置的迭代器，来表示“未找到”或“错误”。
- I/O 流库依赖每个流中的状态来指示错误，而且可能会抛出异常来表示错误发生(如果用户要求这样的话)，参见 10. 6 节和附录 B. 7. 2。
- 一些标准库特性(如 vector、string 和 bitset)通过抛出异常来表示错误。

标准库的一个设计原则就是，所有的特性都要遵循“基本保证”(参见 19. 5. 3 节)。也就是说，即使发生错误，抛出异常，也不会发生资源泄漏(如内存泄漏)以及破坏标准库类的不变式。

B. 2. 1　异常

一些标准库特性通过抛出异常来表示错误：

标准库异常	
bitset	抛出 invalid_argument、out_of_range、overflow_error
dynamic_cast	如果无法进行转换，抛出 bad_cast
iostream	如果异常开启的话，在错误时抛出 ios_base :: failure
new	如果无法分配内存，抛出 bad_alloc
regex	抛出 regex_error
string	抛出 length_erorr、out_of_range
typeid	如果无法获得 type_info，抛出 bad_typeid
vector	抛出 out_of_range

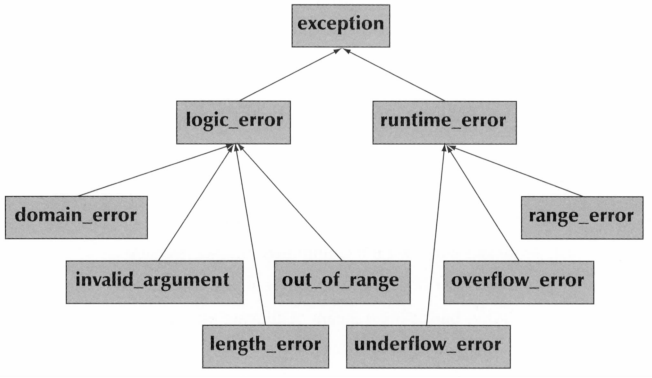
在任何直接或间接使用这些特性的代码中，都可能遇到这些异常。除非你确定你使用这些特性的方式不会产生异常，否则最好保证在某处(比如在 main( )中)捕获标准库异常类层次中的根类(如 exception)，这样就不会遗漏任何异常。

我们强烈建议你不要抛出内置类型，如 int 或 C 风格字符串，应该抛出专门定义用作异常的类型的对象。比如，可以使用从标准库类 exception 派生出的类：

```
class exception {
public:
    exception();
    exception(const exception&);
    exception& operator=(const exception&);
    virtual ~exception();
    virtual const char* what() const;
};
```

函数 what( )可以用来获取一个字符串，这个字符串说明了导致异常的错误是什么。

通过下面的异常分类，我们可以很好地了解标准库异常类的层次关系：

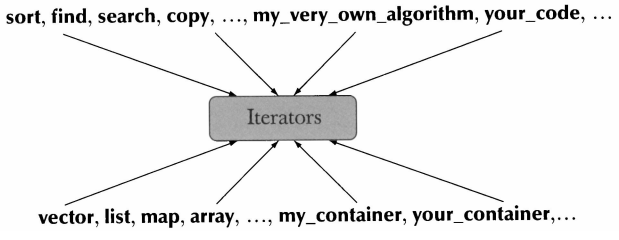


我们可以像下面这样通过派生标准库异常来定义自己的异常类：

```
struct My_error : runtime_error {
        My_error(int x) : interesting_value(x) { }
        int interesting_value;
        const char* what() const { return "My_error"; }
};
```

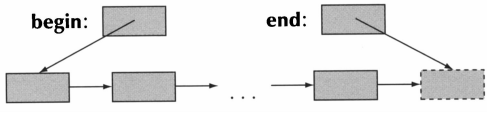
B. 3　迭代器

迭代器是联系标准库算法及其数据的纽带。从相反的角度，你也可以说迭代器是最小化算法和它所处理的数据之间依赖关系的机制（参见 20. 3 节）：



B. 3. 1　迭代器模型

迭代器提供了间接访问数据元素（例如，用 * 解引用）以及移动到新元素（例如，用 ++ 移动到下一元素）的能力，在这些方面它与指针很接近。我们可以用两个迭代器所构成的半开区间［begin: end）来定义一个元素序列：



即 begin 指向序列的第一个元素，而 end 指向序列的最后一个元素之后的位置。因此，不要读写*end。注意，对于空序列 begin == end，即对任意 p，［p: p）都表示空序列。

读取序列内容的一般方法是：使用一对迭代器（b，e），通过 ++ 来遍历序列，直至到达末尾 e：

```
while (b!=e) {    // use != rather than <
        // do something
        ++b;      // go to next element
}
```

在序列中进行搜索的算法通常返回指向序列末尾的迭代器来表示“未找到”(参见 20.3 节),例如:

```
p = find(v.begin(),v.end(),x);      // look for x in v
if (p!=v.end()) {
    // x found at p
}
else {
    // x not found in [v.begin():v.end())
}
```

向序列中写入数据一般只需一个指向首元素的迭代器,保证不超出序列末尾是程序员的责任。例如:

```
template<class Iter> void f(Iter p, int n)
{
    while (n>0) *p++=--n;
}

vector<int> v(10);
f(v.begin(),v.size());      // OK
f(v.begin(),1000);          // big trouble
```

一些标准库实现了范围检查,即对于上面程序中最后一次 f( )调用,会抛出一个异常。但是,当你编写可移植的程序时,不能假定标准库提供了这一功能,有很多实现并不支持范围检查。

我们可以对迭代器进行如下运算:

迭代器运算	
++p	前增:使 p 指向序列中的下一个元素或者超出最后元素之后的一个位置(“前进一个元素”),表达式的值为 p+1
p++	后增:使 p 指向序列中的下一个元素或者超出最后元素之后的一个位置(“前进一个元素”),表达式的值为 p(操作之前的值)
--p	前减:使 p 指向前一个元素(“后退一个元素”),表达式的值为 p-1
p--	后减:使 p 指向前一个元素(“后退一个元素”),表达式的值为 p(操作之前的值)
*p	访问(解引用):*p 引用 p 指向的元素
p[n]	访问(下标):p[n]引用 p+n 指向的元素,等价于*(p+n)
p->m	访问(成员访问):等价于(*p).m
p==q	相等判定:如果 p 和 q 指向相同元素或者都指向最后元素之后的位置,结果为 true
p!=q	不等判定:!(p==q)
p<q	判断 p 指向的元素是否位于 q 指向的元素之前
p<=q	p<q || p==q
p>q	判断 p 指向的元素是否位于 q 指向的元素之后
p>=q	p>q || p==q
p+=n	前进 n 个元素:使 p 指向当前元素之后第 n 个元素
p-=n	后退 n 个元素:使 p 指向当前元素之前第 n 个元素
q=p+n	q 指向 p 所指向的元素之后第 n 个元素
q=p-n	q 指向 p 所指向的元素之前第 n 个元素
advance(p, n)	前进:类似 p+=n,可用于 p 不是随机访问迭代器的情况,它花费 n 个步骤(在列表中移动)来获得最终位置
x=difference(p, q)	距离:类似 q-p,可用于 p 不是随机访问迭代器的情况,它花费 n 个步骤(在列表中移动)来获得两个迭代器的距离

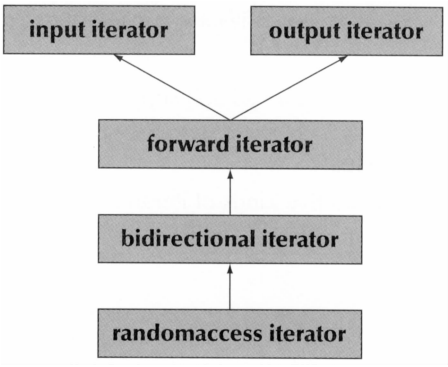
注意,不是每种迭代器(参见附录 B.3.2)都支持所有运算。

B.3.2　迭代器类别

标准库提供 5 种迭代器(5 个“迭代器类别”):

迭代器类别	
输入迭代器	可以使用 ++ 向前移动，只能读取元素——使用* 。可以使用 == 和 != 判断相等性。istream 提供的就是这种迭代器，参见 21.7.2 节
输出迭代器	可以使用 ++ 向前移动，只能写元素——使用* 。ostream 提供的就是这种迭代器，参见 21.7.2 节
前向迭代器	可以反复使用 ++ 向前移动，可以使用* 读写元素（除非元素是 const，const 元素只能读）。如果指向的是类对象，可以使用 -> 访问成员
双向迭代器	既可向前（使用 ++）也可向后（使用 --），可以使用* 读写元素（const 元素只能读）。list、map 和 set 提供的就是这种迭代器
随机访问迭代器	既可向前（使用 ++ 或 +=）也可向后（使用 -- 或 -=），可以使用* 或[ ]读写元素（const 元素只能读）。可以使用下标，可以使用 + 加上一个整数，可以使用 - 减去一个整数。可以使用减法获得指向同一序列的两个随机访问迭代器的距离。可以使用 <、<=、> 和 >= 比较两个随机访问迭代器。vector 提供的就是这种迭代器

逻辑上，这些迭代器构成如下层次（参见 20.8 节）：



注意，由于迭代器类别不是类，因此迭代器层次不是利用类派生实现的类层次。如果你需要深入了解迭代器类别，请查阅一些进阶材料，阅读 iterator_traits 有关内容。

每个容器都提供特定类别的迭代器：

- vector——随机访问迭代器
- list——双向迭代器
- deque——随机访问迭代器
- biset——无
- set——双向迭代器
- multiset——双向迭代器
- map——双向迭代器
- multimap——双向迭代器
- unordered_set——前向迭代器
- unordered_multiset——前向迭代器
- unordered_map——前向迭代器
- unordered_multimap——前向迭代器

B.4　容器

容器是对象序列，序列中元素的类型是称为 value_type 的成员类型。最常用的容器包括：

顺序容器	
array < T, N >	固定大小数组，由 N 个类型为 T 的元素构成(C++0x)
deque < T >	双端队列
list < T >	双向链表
vector < T >	元素类型为 T 的动态数组

关联容器	
map < K, V >	K 到 V 的映射，(K, V)对的序列
multimap < K, V >	K 到 V 的映射，允许重复关键字
set < K >	K 的集合
multiset < K >	K 的集合(允许重复关键字)
unordered_map < K, V >	使用散列函数从 K 映射到 V(C++0x)
unordered_multimap < K, V >	使用散列函数从 K 映射到 V，允许重复关键字(C++0x)
unordered_set < K >	使用散列函数的 K 的集合(C++0x)
unordered_multiset < K >	使用散列函数的 K 的集合，允许重复关键字(C++0x)

容器适配器	
priority_queue < T >	优先队列
queue < T >	队列，支持 push( )和 pop( )操作
stack < T >	栈，支持 push( )和 pop( )操作

这些容器定义于 < vector > 、< list > 等头文件中(参见附录 B. 1. 1)。顺序容器的空间是连续分配的或是链表，元素类型为 value_type(上表中的符号 T)。关联容器是链接结构(树)，节点类型为 value_type(上表中的(K, V)对)。set、map 和 multimap 的序列是按关键字值(K)排序的。unordered 容器的序列不保证顺序。multimap 与 map 的不同之处在于，允许一个关键字值出现多次。容器适配器是在其他容器之上构造而来的，并定义了专门的操作。

如果对选择哪种容器有疑惑，请使用 vector，除非你有充分的理由选用其他容器。

容器使用“分配器”分配和释放内存(参见 19. 3. 6 节)。本附录不讨论分配器，如果需要，请查阅专家级文献。默认情况下，分配器使用 new 和 delete 为其元素申请和释放内存。

一个访问操作一般有两个版本：一个用于 const 对象，另一个版本用于非 const 对象(参见 18. 4 节)。

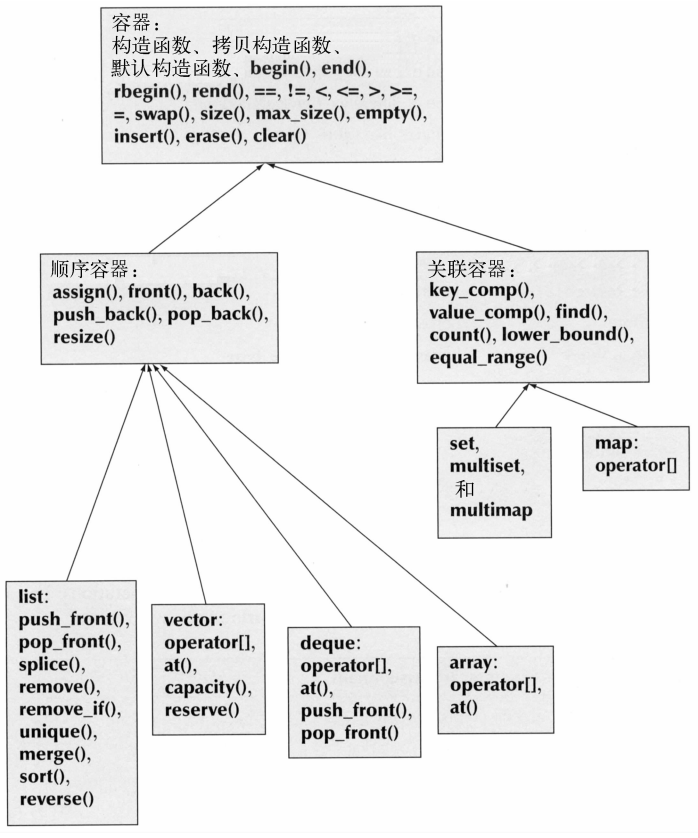
本节列出的成员，都是标准容器共有的或者几乎是共有的，更多细节请参考第 20 章。某个特定容器特有的成员，如 list 的 splice( )，并不在本节讨论范围内，这方面内容请查阅专家级书籍。

某些数据类型提供了标准容器所能提供的大部分功能，但并不是全部。我们有时称这些数据类型为“近似容器”，一些最常用的近似容器如下表所示：

“近似容器”	
T[n](内置数组)	不具备 size( )和其他成员函数，如果可能的话，尽量使用 vector、string 或 array 等容器，而不要使用内置数组
string	只包含字符，但提供了有用的文本处理操作，如连接(+ 和 +=)，应尽量使用标准库 string，而不要使用其他类型的字符串
valarray	数值向量，支持向量操作，但为了提高性能，有很多限制；除非你需要做大量向量算术运算，否则不要使用它

B. 4. 1　容器操作概述

标准容器提供的操作可总结如下：



B.4.2　成员类型

容器可以定义一组成员类型：

成员类型	
value_type	元素类型
size_type	下标、元素数量等的类型
difference_type	迭代器距离的类型
iterator	与 value_type* 类似
const_iterator	与 const value_type* 类似
reverse_iterator	与 value_type* 类似
const_reverse_iterator	与 const value_type* 类似
reference	value_type&
const_reference	const value_type&
pointer	与 value_type* 类似
const_pointer	与 const value_type* 类似
key_type	关键字类型(只有关联容器才有)
mapped_type	映射值类型(只有关联容器才有)
key_compare	比较标准的类型(只有关联容器才有)
allocator_type	内存管理器类型

B.4.3　构造函数、析构函数和赋值

容器提供了多种构造函数和赋值操作。对于一个称为 C 的容器(如 vector < double > 或 map < string, int >)，可使用如下操作：

构造函数、析构函数和赋值	
C c;	c 为空容器
C()	创建一个空容器
C c(n);	c 被初始化为 n 个元素的容器，元素被赋予默认值(关联容器不适用)
C c(n, x);	c 被初始化为包含 x 的 n 个副本(关联容器不适用)
C c(b, e);	c 被初始化为包含[b:e)之间的元素
C c(c2);	c 被初始化为容器 c2 的副本
~C()	销毁容器及其所有元素(通常被隐式调用)
c1 = c2	拷贝赋值，将 c2 的所有元素复制到 c1 中，因此，赋值完成后 c1 == c2
c. assign(n, x)	将 x 的 n 个副本赋予 c
c. assign(b, e)	将[b:e)之间的元素赋予 c

注意，对一些容器和一些元素类型，构造函数或元素复制可能会抛出异常。

B. 4. 4　迭代器

一个容器可以看做按其迭代器定义的顺序，或按相反顺序排列的元素序列。关联容器的顺序则是由比较操作(默认比较操作是运算符 <)决定的。

迭代器	
p = c. begin()	p 指向 c 的首元素
p = c. end()	p 指向 c 的尾元素之后的位置
p = c. rbegin()	p 指向 c 的逆序的首元素
p = c. rend()	p 指向 c 的逆序的尾元素之后的位置

B. 4. 5　元素访问

一些元素可以直接访问：

元素访问	
c. front()	c 的首元素的引用
c. back()	c 的尾元素的引用
c[i]	c 的第 i 个元素的引用，不做范围检查(list 不适用)
c. at(i)	c 的第 i 个元素的引用，有范围检查(只适用于 vector 和 deque)

一些标准库实现总是进行范围检查，特别是在调试状态下。但如果你想编写可移植的代码，不能假定标准库会做范围检查，以此来保证正确性，同样也不能假定标准库不会做类型检查，以此获得好的性能。如果这很重要，请认真检查你的代码。

B. 4. 6　栈和队列操作

标准库 vector 和 deque 提供了在序列末端的高效操作。另外，list 和 deque 提供了在序列首的高效操作。

栈和队列操作	
c. push_back(x)	将 x 添加到 c 的末尾
c. pop_back()	删除 c 的尾元素
c. push_front(x)	将 x 添加到 c 的首元素之前(只适用于 list 和 deque)
c. pop_front()	删除 c 的首元素(只适用于 list 和 deque)

注意，push_front()和 push_back()向容器中拷贝一个元素。这意味着容器规模增加 1。如果元素类型的拷贝构造函数能抛出异常，压栈操作可能会失败。

注意，弹出栈操作不返回值。如果这些操作返回值，而且元素类型的拷贝构造函数能抛出异

常，则可能会使实现变得非常复杂。访问栈和队列元素请使用 front( ) 和 back( )（参见附录 B. 4. 5）。本小节中没有给出每个操作的完整的前提要求，你可以猜测（如果猜错，编译器会告知你）或者查阅更详细的文档。

B. 4. 7　列表操作

容器还提供了列表操作：

列表操作	
q = c. insert(p, x)	将 x 添加到 p 之前
q = c. insert(p, n, x)	将 x 的 n 份副本添加到 p 之前
q = c. insert(p, first, last)	将[first: last) 之间的原书添加到 p 之前
q = c. erase(p)	将 c 中位置 p 处的元素删除
q = c. erase(first, last)	删除 c 中[first. last) 之间的元素
c. clear( )	删除 c 中所有元素

对于 insert( ) 函数，结果 q 指向插入的最后一个元素。对于 earse( ) 函数，q 指向最后一个被删除元素之后的元素。

B. 4. 8　大小和容量

大小是指容器中的元素个数，容量是指容器在不扩充内存的情况下所能容纳的最大元素数：

大小和容量	
x = c. size( )	x 为 c 的元素数目
c. empty( )	c 是否为空
x = c. max_size( )	x 为 c 能容纳的最大元素数
x = c. capacity( )	x 是为 c 分配的内存空间大小（仅适用于 vector 和 string）
c. reserve(n)	为 c 留出 n 个元素的空间（仅适用于 vector 和 string）
c. resize(n)	将 c 的大小改变为 n（仅适用于 vector、string、list 和 deque）

当改变大小或容量时，元素可能被移动到新的内存位置。这意味着指向元素的迭代器（以及指针和引用）将变为无效（它们仍指向旧的元素存储位置）。

B. 4. 9　其他操作

容器可以拷贝（参见附录 B. 4. 3）、比较以及交换：

比较和交换	
c1 == c2	c1 和 c2 的所有对应元素是否都相等
c1 ! = c2	c1 和 c2 的某个对应元素是否不相等
c1 < c2	c1 字典序是否在前
c1 < = c2	c1 字典序在 c2 之前，或与 c2 相同
c1 > c2	c1 字典序是否在后
c1 > = c2	c1 字典序在 c2 之后，或与 c2 相同
swap(c1, c2)	交换 c1 和 c2 的元素
c1. swap(c2)	交换 c1 和 c2 的元素

当用一个运算符（如 < ）比较两个容器时，实际是用等价的元素运算符（即 < ）来比较对应的元素。

B. 4. 10　关联容器操作

关联容器提供了基于关键字的查找：

关联容器操作	
c[k]	引用关键字为 k 的元素(适用于关键字唯一的容器)
p = c. find(k)	p 指向第一个关键字为 k 的元素
p = c. lower_bound(k)	p 指向第一个关键字为 k 的元素
p = c. upper_bound(k)	p 指向第一个关键字大于 k 的元素
pair(p1, p2) = c. equal_range(k)	[p1, p2)为关键字为 k 的所有元素
r = c. key_comp()	r 为关键字比较对象(比较函数)的副本
r = c. value_comp()	r 为映射值比较对象(比较函数)的副本。如果关键字未找到,则返回 c. end()

equal_range 返回的 pair 中的第一个迭代器为 lower_bound,第二个迭代器为 upper_bound。如果你希望打印 multimap < string, int > 中所有关键字值为"Marian"的元素,可以这样做:

```
string k = "Marian";
typedef multimap<string,int>::iterator MI;
pair<MI,MI> pp = m.equal_range(k);
if (pp.first!=pp.second)
    cout << "elements with value '" << k << "':\n";
else
    cout << "no element with value '" << k << "'\n";
for (MI p = pp.first; p!=pp.second; ++p) cout << p->second << '\n';
```

其中对 equal_range 的调用等价于:

```
pair<MI,MI> pp = make_pair(m.lower_bound(k),m.upper_bound(k));
```

但是,第二种方式的执行时间可能是 equal_range 的两倍。有序序列也提供了 equal_range、lower_bound 和 upper_bound 算法(参见附录 B. 5. 4)。pair 的定义参见附录 B. 6. 3。

B. 5　算法

< algorithm > 定义了大约 60 个标准算法。所有这些算法都是对一对迭代器所标定的序列(输入)或者一个单一迭代器(输出)进行操作。

当对两个序列进行复制、比较等操作时,第一个序列用一对迭代器[b:e)表示,但第二个序列只用一个单一迭代器 b2 表示。因为第一个序列已经给出了元素个数,第二个序列只给出起始位置就已经足够了,例如,第二个序列的元素数目应该与第一个序列相同——[b2:b2 + (e - b))。

某些算法,例如 sort,要求随机访问迭代器,而其他很多算法,如 find,只顺序读取元素,因此只需一个向前迭代器就能工作。

很多算法遵循这样一个常用约定:通过返回序列尾来表示"未找到",我们不再对每个算法重复这一点。

B. 5. 1　非修改型序列算法

非修改型算法只读取序列元素,但不重排元素顺序,也不修改元素的值。

非修改型序列算法	
f = for_each(b, e, f)	对[b:e)中的每个元素执行 f,返回 f
p = find(b, e, v)	p 指向[b:e)中第一次出现 v 的位置
p = find_if(b, e, f)	p 指向[b:e)中第一个使 f(*p)为真的元素
p = find_first_of(b, e, b2, e2)	p 指向[b:e)中第一个满足*p == *q 的元素,q 为[b2:e2)中位置
p = find_first_of(b, e, b2, e2, f)	p 指向[b:e)中第一个使 f(*p, *q)为真的元素,q 为[b2:e2)中对应位置
p = adjacent_find(b, e)	p 指向[b:e)中第一个满足*p == *(p + 1)的元素
p = adjacent_find(b, e, f)	p 指向[b:e)中第一个使 f(*p, *(p + 1))为真的元素
equal(b, e, b2)	[b:e)中所有元素都和[b2:b2 + (e - b))中对位元素相等吗

（续）

非修改型序列算法	
equal(b, e, b2, f)	对[b:e)和[b2:b2+(e-b))中所有对位元素*p、*q，f(*p, *q)都为真吗
pair(p1, p2) = mismatch(b, e, b2)	(p1, p2)指向[b:e)和[b2:b2+(e-b))中第一对不等的对位元素，即！(*p1 == *p2)
pair(p1, p2) = mismatch(b, e, b2, f)	(p1, p2)指向[b:e)和[b2:b2+(e-b))中第一对满足！f(*p1, *p2)的对位元素
p = search(b, e, b2, e2)	p 指向[b:e)中第一个在[b2:e2)中有相等元素的元素
p = search(b, e, b2, e2, f)	p 指向[b:e)中第一个这样的元素：在[b2:e2)中存在一个元素*q，满足 f(*p, *q)
p = find_end(b, e, b2, e2)	p 指向[b:e)中最后一个在[b2:e2)中有相等元素的元素
p = find_end(b, e, b2, e2, f)	p 指向[b:e)中最后一个这样的元素：在[b2:e2)中存在一个元素*q，满足 f(*p, *q)
p = search_n(b, e, n, v)	p 指向[b:e)中第一个满足下面条件的区间[p:p+n)的第一个元素：其全部元素都等于 v
p = search_n(b, e, n, v, f)	p 指向[b:e)中第一个全部元素都满足 f(*p, v)的区间[p:p+n)的第一个元素
x = count(b, e, v)	x 为[b:e)中 v 出现的次数
x = count_if(b, e, v, f)	x 为[b:e)中满足 f(*p, v)的元素的数目

注意，标准库并不会阻止将一个修改元素的操作传递给 for_each，这是可接受的。但对于其他一些算法（如 count 或 ==），则不接受修改元素的操作。

下面是一个（正确使用的）例子：

```
bool odd(int x) { return x&1; }

int n_even(const vector<int>& v)     // count the number of even values in v
{
        return v.size()-count_if(v.begin(),v.end(),odd);
}
```

B.5.2　修改型序列算法

修改型算法（也称为改变型序列算法）可以（通常也确实是）修改参数序列中的元素。

修改型序列算法	
p = transform(b, e, out, f)	对[b:e)中每个*p1 执行*p2 = f(*p1)，将结果 *p2 写到[out:out+(e-b))中对应位置；p = out+(e-b)
p = transform(b, e, b2, out, f)	对[b:e)和[b2:b2+(e-b))中每对对位元素*p1 和*p2 执行*p3 = f(*p1, *p2)，将结果*p2 写到[out:out+(e-b))中对应位置；p = out+(e-b)
p = copy(b, e, out)	将[b:e)拷贝至[out:p)
p = copy_backward(b, e, out)	从最后一个元素开始将[b:e)拷贝至[out:p)
p = unique(b, e)	移动[b:e)中的元素，使得[b:p)中相邻元素不重复（“重复”由 == 判定）
p = unique(b, e, f)	移动[b:e)中的元素，使得[b:p)中相邻元素不重复（“重复”由 f 判定）
p = unique_copy(b, e, out)	将[b:e)拷贝至[out:p)，不拷贝相邻的重复元素
p = unique_copy(b, e, out, f)	将[b:e)拷贝至[out:p)，不拷贝相邻的重复元素（“重复”由 f 判定）
replace(b, e, v, v2)	将[b:e)中的所有值为 v 的元素用 v2 替换
replace(b, e, f, v2)	将[b:e)中的所有满足 f(*q)的元素用 v2 替换
p = replace_copy(b, e, out, v, v2)	将[b:e)拷贝至[out:p)，其中所有值为 v 的元素用 v2 替换
p = replace_copy(b, e, out, f, v2)	将[b:e)拷贝至[out:p)，其中所有满足 f(*q)的元素用 v2 替换
p = remove(b, e, v)	移动[b:e)中的元素，使得[b:p)中不出现 v
p = remove(b, e, v, f)	移动[b:e)中的元素，使得[b:p)中无满足 f(*q)的元素
p = remove_copy(b, e, out, v)	将[b:e)中值不等于 v 的元素拷贝至[out:p)

（续）

修改型序列算法	
p = remove_copy_if(b, e, out, f)	将[b:e)中不满足 f(*q)的元素拷贝至[out:p)
reverse(b, e)	倒转[b:e)中元素的顺序
p = reverse_copy(b, e, out)	将[b:e)中元素按逆序拷贝至[out:p)
rotate(b, m, e)	元素循环移位：将[b:e)看做一个圈——首元素在尾元素之后。将*b 移动到*m，其他元素依此类推——将*(b+i)移动到*((b+(i+(e-m)%(e-b))
p = rotate_copy(b, m, e, out)	将[b:e)拷贝至[out:p)，元素循环右移 m 位
random_shuffle(b, e)	随机混洗：利用默认的均匀分布随机数发生器将[b:e)中的元素重新排列
random_shuffle(b, e, f)	随机混洗：用 f 作为随机数发生器

shuffle 算法充分混洗序列，就像我们洗牌一样。也就是说，混洗之后，元素是随机排列的，“随机”是由随机数发生器所产生的分布决定的。

请注意，这些算法并不知道其参数是否是一个容器，因而它们并没有添加或删除元素的能力。因此，像 remove 这样的算法不会删除元素、缩短序列，而是将保留下来的元素移动到序列的前端：

```
typedef vector<int>::iterator VII;

void print_digits(const string& s, VII b, VII e)
{
        cout << s;
        while (b!=e) { cout << *b; ++b; }
        cout << '\n';
}
void ff()
{
        int a[] = { 1,1,1, 2,2, 3, 4,4,4, 3,3,3, 5,5,5,5, 1,1,1 };
        vector<int> v(a,a+sizeof(a)/sizeof(int));
        print_digits("all: ",v.begin(), v.end());

        vector<int>::iterator pp = unique(v.begin(),v.end());
        print_digits("head: ",v.begin(),pp);
        print_digits("tail: ",pp,v.end());

        pp=remove(v.begin(),pp,4);
        print_digits("head: ",v.begin(),pp);
        print_digits("tail: ",pp,v.end());
}
```

输出结果如下：

```
all:   11122344433335555111
        head: 1234351
        tail:   443335555111
        head: 123351
        tail:   1443335555111
```

B.5.3　工具算法

从技术角度看，这些工具算法也都是修改型序列算法，但我们觉得将它们单独列出更好，以免被忽略。

工具算法	
swap(x, y)	交换 x 和 y
iter_swap(p, q)	交换*p 和*q
swap_ranges(b, e, b2)	交换[b:e)和[b2:b2+(e-b))的元素

（续）

工具算法	
fill(b, e, v)	将 v 赋予[b:e)中的每个元素
fill_n(b, n, v)	将 v 赋予[b:b+n)中的每个元素
generate(b, e, f)	将 f( )赋予[b:e)中的每个元素
generate_n(b, n, f)	将 f( )赋予[b:e)中的每个元素
uninitialized_fill(b, e, v)	将[b:e)中所有元素初始化为 v
uninitialized_copy(b, e, out)	用[b:e)中的元素初始化[out:out+(e-b))中对应的元素

注意，未初始化的序列只能出现在最底层的程序中，通常是在容器的实现代码中。uninitialized_fill 和 uninitialized_copy 处理的元素必须是内置类型或者是未初始化的。

B.5.4　排序和搜索

排序和搜索是非常基础的操作，而程序员对其的要求可能差别很大。默认情况下，比较操作通过 < 运算符来完成，而元素 a 和 b 的值相等通过!(a<b)&&!(b<a)来判定，而不是使用 == 运算符。

排序和搜索	
sort(b, e)	排序[b:e)
sort(b, e, f)	排序[b:e)，使用 f(*p, *q)进行比较操作
stable_sort(b, e)	排序[b:e)，保持相等元素的原有顺序
stable_sort(b, e, f)	排序[b:e)，使用 f(*p, *q)进行比较操作，保持相等元素的原有顺序
paritial_sort(b, m, e)	排序[b:e)，保证[b:m)有序，[m:e)可以不必有序
partial_sort(b, m, e, f)	排序[b:e)，使用 f(*p, *q)进行比较操作，保证[b:m)有序，[m:e)可以不必有序
partial_sort_copy(b, e, b2, e2)	排序[b:e)，保证有足够多(e2-b2 个)的有序元素复制到[b2:e2)即可
partial_sort_copy(b, e, b2, e2, f)	排序[b:e)，使用 f 作为比较操作，保证有足够多的有序元素复制到[b2:e2)即可
nth_element(b, e)	将[b:e)中排在第 n 位的元素移动到正确位置
nth_element(b, e, f)	将[b:e)中排在第 n 位的元素移动到正确位置，使用 f 作为比较操作
p=lower_bound(b, e, v)	p 指向[b:e)中 v 第一次出现的位置
p=lower_bound(b, e, v, f)	p 指向[b:e)中 v 第一次出现的位置，使用 f 作为比较操作
p=upper_bound(b, e, v)	p 指向[b:e)中第一个大于 v 的元素
p=upper_bound(b, e, v, f)	p 指向[b:e)中第一个大于 v 的元素，使用 f 作为比较操作
binary_search(b, e, v)	有序序列[b:e)中包含 v 吗
binary_search(b, e, v, f)	有序序列[b:e)中包含 v 吗？f 作为比较操作
pair(p1, p2)=equal_range(b, e, v)	[p1:p2)为[b:e)的子序列，其中元素均为 v，大致是在序列中二分搜索 v
pair(p1, p2)=equal_range(b, e, v, f)	[p1:p2)为[b:e)的子序列，其中元素均为 v，f 作为比较操作，大致是在序列中二分搜索 v
p=merge(b, e, b2, e2, out)	将有序序列[b2:e2)和[b:e)合并为一个有序序列，结果存入[out:p)
p=merge(b, e, b2, e2, out, f)	将有序序列[b2:e2)和[b:e)合并为一个有序序列，结果存入[out,out+p)，f 作为比较操作
inplace_merge(b, m, e)	原址合并，将有序序列[b:m)和[m:e)合并为一个有序序列[b:e)
inplace_merge(b, m, e, f)	原址合并，f 作为比较操作
p=paritition(b, e, f)	将满足 f(*p1)的元素置于[b:p)，其他元素置于[p:e)
p=stable_partition(b, e, f)	将满足 f(*p1)的元素置于[b:p)，其他元素置于[p:e)，保持相对顺序

例如：

```
vector<int> v;
list<double> lst;
v.push_back(3); v.push_back(1);
v.push_back(4); v.push_back(2);
lst.push_back(0.5); lst.push_back(1.5);
lst.push_back(2); lst.push_back(2.5);      // lst is in order
sort(v.begin(),v.end());                   // put v in order
vector<double> v2;
merge(v.begin(),v.end(),lst.begin(),lst.end(),back_inserter(v2));
for (int i = 0; i<v2.size(); ++i) cout << v2[i] << ", ";
```

关于 inserter，参见附录 B. 6. 1。输出结果为：

```
0.5, 1, 1.5, 2, 2, 2.5, 3, 4,
```

equal_range、lower_bound 和 upper_bound 的使用方法与关联容器的对应版本相同，参见附录 B. 4. 10。

B. 5. 5　集合算法

这些算法将序列视为元素集合，提供了基本的集合操作。这些算法假定输入序列是有序的，而它们生成的输出也是有序的：

集合算法	
includes(b, e, b2, e2)	[b2:e2)中所有元素是否都在[b:e)中
includes(b, e, b2, e2, f)	[b2:e2)中所有元素都在[b:e)中？f 作为比较操作
p = set_union(b, e, b2, e2, out)	构造有序序列[out:p)，包含[b:e)或[b2:e2)中所有元素
p = set_union(b, e, b2, e2, out, f)	构造有序序列[out:p)，包含[b:e)和[b2:e2)中所有元素，f 作为比较操作
p = set_intersection(b, e, b2, e2, out)	构造有序序列[out:p)，包含[b:e)和[b2:e2)中同时出现的元素
p = set_intersection(b, e, b2, e2, out, f)	构造有序序列[out:p)，包含[b:e)和[b2:e2)中同时出现的元素，f 作为比较操作
p = set_difference(b, e, b2, e2, out)	构造有序序列[out:p)，包含在[b:e)中出现但在[b2:e2)中没有出现的元素
p = set_difference(b, e, b2, e2, out, f)	构造有序序列[out:p)，包含在[b:e)中出现但在[b2:e2)中没有出现的元素，f 作为比较操作
p = set_symmetric_difference(b, e, b2, e2, out)	构造有序序列[out:p)，包含[b:e)中或[b2:e2)中出现的但不同时在两者中出现的元素
p = set_symmetric_difference(b, e, b2, e2, out, f)	构造有序序列[out:p)，包含在[b:e)中出现或[b2:e2)中出现但不同时在两者中出现的元素，f 作为比较操作

B. 5. 6　堆

堆是这样一种数据结构：它将值最大的元素保存在最前面。堆算法允许程序员将一个随机访问序列作为一个堆来处理：

堆操作	
make_heap(b, e)	将序列[b:e)整理为一个堆
make_heap(b, e, f)	将序列[b:e)整理为一个堆，f 作为比较操作
push_heap(b, e)	将元素加入堆(中正确的位置)
push_heap(b, e, f)	将元素加入堆，f 作为比较操作
pop_heap(b, e)	将最大元素从堆中删除
pop_heap(b, e, f)	将最大元素从堆中删除，f 作为比较操作
sort_heap(b, e)	堆排序
sort_heap(b, e, f)	堆排序，f 作为比较操作

堆的目标是提供快速的元素添加和最大元素访问操作，其主要用途是实现优先队列。

B.5.7　排列

排列用于生成序列元素的组合。例如，abc 的排列有 abc、acb、bac、bca、cab 和 cba。

排列	
x = next_permutation(b, e)	生成字典序中[b:e)的下一个排列
x = next_permutation(b, e, f)	生成字典序中[b:e)的下一个排列，f 作为比较操作
x = prev_permutation(b, e)	生成字典序中[b:e)的前一个排列
x = prev_permutation(b, e, f)	生成字典序中[b:e)的前一个排列，f 作为比较操作

如果[b:e)当前的排列已经是最后一个排列(如上例中的 cba)，next_permutation 的返回值(x)为 false，在这种情况下，返回第一个排列(如上例中的 abc)。如果[b:e)当前的排列已经是第一个排列(如上例中的 abc)，prev_permutation 的返回值(x)为 false，在这种情况下，返回最后一个排列(如上例中的 cba)。

B.5.8　min 和 max

比较操作在很多场合下都是很有用的：

min 和 max	
x = max(a, b)	x 为 a 和 b 中较大者
x = max(a, b, f)	x 为 a 和 b 中较大者，f 作为比较操作
x = min(a, b)	x 为 a 和 b 中较小者
x = min(a, b, f)	x 为 a 和 b 中较小者，f 作为比较操作
p = max_element(b, e)	p 指向[b:e)中最大元素
p = max_element(b, e, f)	p 指向[b:e)中最大元素，f 作为元素比较操作
p = min_element(b, e)	p 指向[b:e)中最小元素
p = min_element(b, e, f)	p 指向[b:e)中最小元素，f 作为元素比较操作
lexicographical_compare(b, e, b2, e2)	[b:e) < [b2:e2)吗
lexicographical_compare(b, e, b2, e2, f)	[b:e) < [b2:e2)吗？f 作为元素比较操作

B.6　STL 工具

STL 提供了一些工具，能方便使用 STL 算法。

B.6.1　插入器

一些算法通过迭代器将结果输出到容器中，这意味着迭代器指向的元素和之后一个元素可以被改写。这也意味着可能出现溢出，从而导致内存错误。例如：

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(), 200,7 );    // assign 7 to vi[0]..[199]
}
```

如果 vi 中元素数少于 200，我们就有麻烦了。

在 < iterator > 中，标准库提供了三种迭代器来解决问题，它们并不改写旧元素，而是向容器中加入(插入)新元素。标准库提供了三个函数来生成这些插入迭代器：

插入器	
r = back_inserter(c)	*r = x，引起一次 c.push_back(x)操作
r = front_inserter(c)	*r = x，引起一次 c.push_front(x)操作
r = inserter(c, p)	*r = x，引起一次 c.insert(p, x)操作

对于 inserter(c, p), p 必须是容器 c 的一个合法迭代器。当用插入迭代器将一个值写入容器时,容器的规模增长一个元素。插入器通过 push_back( )、push_front( )或 insert( )向容器插入一个新元素,而不是改写已有元素,例如:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi), 200,7 );   // add 200 7s to the end of vi
}
```

B. 6. 2　函数对象

很多标准库算法都接受函数对象(或函数)作为参数,这允许程序员控制其工作方式。函数对象的常见用途是比较操作、断言(返回布尔值的函数)和算术运算。在 < functional > 中,标准库提供了一些常用的函数对象。

断言	
p = equal_to < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x == y
p = not_equal_to < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x! = y
p = greater < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x > y
p = less < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x < y
p = greater_equal < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x >= y
p = less_equal < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x <= y
p = logical_and < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x&&y
p = logical_or < T > ( )	当 x 和 y 是类型 T 时, p(x, y)表示 x||y
p = logical_not < T > ( )	当 x 是类型 T 时, p(x)表示! x

例如:

```
vector<int> v;
// . . .
sort(v.begin(),v.end(),greater<int>());      // sort v in decreasing order
```

注意, logical_and 和 logical_or 总是对两个参数都进行求值(&& 和||则不是这样,而是采取短路求值策略)。

算术运算	
f = plus < T > ( )	当 x 和 y 是类型 T 时, f(x, y)表示 x + y
f = minus < T > ( )	当 x 和 y 是类型 T 时, f(x, y)表示 x - y
f = multiplies < T > ( )	当 x 和 y 是类型 T 时, f(x, y)表示 x * y
f = divides < T > ( )	当 x 和 y 是类型 T 时, f(x, y)表示 x/y
f = modulus < T > ( )	当 x 和 y 是类型 T 时, f(x, y)表示 x% y
f = negate < T > ( )	当 x 是类型 T 时, f(x, y)表示 - x
适配器	
f = bind2nd(g, y)	f(x)表示 g(x, y)
f = bind1st(g, x)	f(y)表示 g(x, y)
f = mem_fun(mf)	f(p)表示 p -> mf( )
f = mem_fun_ref(mf)	f(r)表示 r. mf( )
f = not1(g)	f(x)表示! g(x)
f = not2(g)	f(x, y)表示! g(x, y)

B. 6. 3　pair

在 < utility > 中,标准库提供了一些“工具组件”, pair 就是其中之一:

```
template <class T1, class T2>
    struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;
        pair();  // default constructor
        pair(const T1& x , const T2& y );

        // copy operations:
        template<class U , class V > pair(const pair<U , V >& p );
    };

template <class T1, class T2>
    pair<T1,T2> make_pair(T1 x, T2 y) { return pair<T1,T2>(x,y); }
```

函数 make_pair( )使 pair 的使用变得简单。例如，下面的函数返回一个值和一个错误指示器：

```
pair<double,error_indicator> my_fct(double d)
{
    errno = 0;     // clear C-style global error indicator
    // do a lot of computation involving d computing x
    error_indicator ee = errno;
    errno = 0;     // clear C-style global error indicator
    return make_pair(x,ee);
}
```

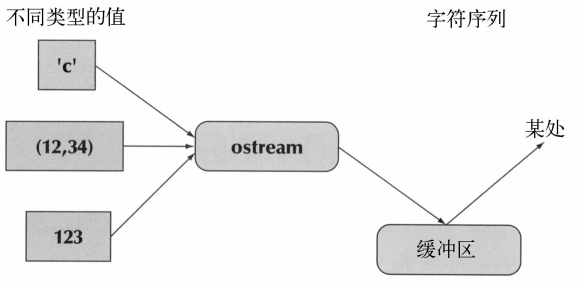
下面是一种常见的用法：

```
pair<int,error_indicator> res = my_fct(123.456);
if (res.second==0) {
    // use res.first
}
else {
    // oops: error
}
```

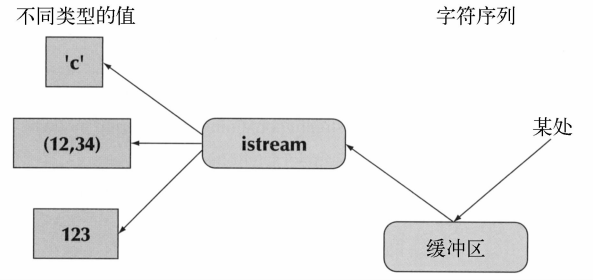
B.7　I/O 流

I/O 流库提供了文本和数值的格式化和未格式化的缓冲 I/O 功能。I/O 流特性在 < istream > 、< ostream > 等头文件中定义，参见附录 B.1.1。

ostream 可以将有类型的对象转换为字符(字节)流：



istream 可以将字符(字节)序列转换为有类型的对象，如下图所示。一个 iostream 既可以像 istream 一样工作，也可以像 ostream 一样工作。上图中的缓冲区是“流缓冲区”(streambuf 对象)。如果你需要将 iostream 映射到一种新设备、文件或内存，请查阅专家级书籍来获得 streambuf 的更详细的内容。

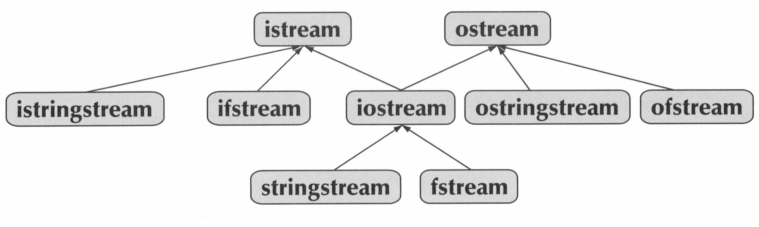


STL 中提供了三种标准流：

标准 I/O 流	
cout	标准字符输出(通常默认是显示器)
cin	标准字符输入(通常默认是键盘)
cerr	标准字符错误输出(未缓冲的)

B.7.1　I/O 流层次

一个 istream 可以连接至一个输入设备(如键盘)、一个文件或者一个 string。类似地，一个 ostream 可以连接至一个输出设备(如文本窗口)、一个文件或者一个 string。I/O 流特性组织为一个类层次：



我们可以通过构造函数或 open( )调用来打开一个流：

流类型	
stringstream(m)	以模式 m 创建一个空的字符流
stringstream(s, m)	以模式 m 创建一个包含字符串 s 的字符串流
fstream( )	创建一个文件流，稍后可以打开
fstream(s, m)	以模式 m 打开名为 s 的文件，创建一个文件流指向此文件
fs. open(s, m)	以模式 m 打开名为 s 的文件，令 fs 指向它
fs. is_open( )	fs 是否已经打开

对于文件流，文件名是 C 风格字符串。

你可以以一种或多种模式打开一个文件：

流模式	
ios_base :: app	追加模式(即添加到文件末尾)
ios_base :: ate	“文件尾”模式(打开文件，并定位到文件尾)
ios_base :: binary	二进制模式——要小心系统相关的行为
ios_base :: in	读模式
ios_base :: out	写模式
ios_base :: trunc	将文件长度截断为 0

对于每种模式，确切的效果依赖于操作系统。如果操作系统不允许以某种特定方式打开文件，则流会进入非 good( )状态。

下面是一个例子：

```
void my_code(ostream& os);      // my code can use any ostream

ostringstream os;               // o for "output"
ofstream of("my_file");
if (!of) error("couldn't open 'my_file' for writing");
my_code(os);                    // use a string
my_code(of);                    // use a file
```

请参见 11.3 节。

B.7.2　I/O 错误处理

iostream 可以处于下列四种状态之一：

流状态	
good( )	操作成功
eof( )	到达输入结尾(“文件尾”)
fail( )	发生了不期望的事情(例如，寻找数字却遇到‘x’)
bad( )	发生了不期望的严重问题(例如，磁盘读错误)

程序员可以通过使用 s.exceptions( )，来要求 iostream 在从 good( )状态转到其他状态时抛出一个异常(参见 10.6 节)。

如果流处于非 good( )状态，试图对其进行操作将不会有任何效果，即“无操作”。

iostream 可以作为条件来使用——如果流状态为 good( )则条件为真(成功)。这样，读取流的程序通常就可以这样编写：

```
X x;  // an "input buffer" for holding one value of type X
while (cin>>x) {
    // do something with x
}
// we get here when >> couldn't read another X from cin
```

B.7.3　输入操作

除了字符串流的输入操作定义于 <string> 之外，其他流输入操作都定义于 <istream>：

格式化输入	
in >> x	根据 x 的类型，从 in 中读取数据存入 x
getline(in, s)	从 in 中读取一行，存入字符串 s

除非特别指出，否则 istream 操作都会返回指向其 istream 的引用，因此我们可以将操作“链接”起来，如 cin >> x >> y;。

未格式化输入	
x = in.get( )	从 in 中读取一个字符，返回其整数值
in.get(c)	从 in 中读取一个字符，存入 c
in.get(p, n)	从 in 中读取最多 n 个字符，存入以 p 开始的数组
in.get(p, n, t)	从 in 中读取最多 n 个字符，存入以 p 开始的数组，t 作为结束符
in.getline(p, n)	从 in 中读取最多 n 个字符，存入以 p 开始的数组，从 in 中删除结束符
in.getline(p, n, t)	从 in 中读取最多 n 个字符，存入以 p 开始的数组，t 作为结束符，从 in 中删除结束符
in.read(p, n)	从 in 中读取最多 n 个字符，存入以 p 开始的数组
x = in.gcount( )	x 为 in 的最后一次未格式化输入操作所读取的字符数

函数 get( )和 getline( )在字符序列末尾放置一个 0；如果读取到结束符(t)，getline( )将其从输入流中删除，而 get( )则不会被删除。read(p, n)不会在字符序列末尾写入一个 0。显然，与非格式化输入操作相比，格式化输入操作更容易使用，也更不容易出错。

B.7.4　输出操作

除了字符串流的输出操作定义于 < string > 之外，其他流输出操作都定义于 < istream > ：

输出操作	
out << x	根据 x 的类型，将 x 写入 out
out. put(c)	将字符 c 写入 out
out. write(p, n)	将字符 p[0]. . p[n -1]写入 out

除非特别指出，否则 ostream 操作都会返回指向其 ostream 的引用，因此我们可以将操作“链接”起来，如 cout << x << y; 。

B.7.5　格式化

流 I/O 的格式由对象类型、流状态、本地化信息(参见 < locale >) 及显式操作共同控制。第 10 章和第 11 章对此进行了详细介绍，因此本节只是列出标准的格式操纵符(改变流状态的操作) ，因为这些操纵符提供了最直接的改变格式的方法。

本地化的相关内容已经超出了本书讨论的范围。

B.7.6　标准格式操纵符

标准库提供了一些操纵符，与不同的格式状态和状态改变相对应。标准操纵符定义于 < ios > 、< istream > 、< ostream > 、< iostream > 和 < iomanip > (接受参数的操纵符) 中：

I/O 操纵符	
s << boolalpha	使用 true 和 false 的符号表示(输入和输出)
s << noboolalpha	s. unsetf(ios_base :: boolalpha)
s << showbase	八进制输出加前缀 0，十六进制加前缀 0x
s << noshowbase	s. unsetf(ios_base :: showbase)
s << showpoint	总是显示小数点
s << noshowpoint	s. unsetf(ios_base :: showpoint)
s << showpos	对于正数显示 +
s << nowhowpos	s. unsetf(ios_base :: showpos)
s >> skipws	跳过空白符
s >> noskipws	s. unsetf(ios_base :: skipws)
s >> uppercase	在数值输出中使用大写字母，如 1. 2E10 和 0X1A2，而不是 1. 2e10 和 0x1a2
s >> nouppercase	使用 x 和 e 而不是 X 和 E
x << internal	在格式模式中指定的位置进行填充
x << left	在值之后填充
x << right	在值之前填充
s << dec	整数基底设置为 10
s << hex	整数基底设置为 16
s << oct	整数基底设置为 8
s << fixed	采用浮点格式 dddd. dd
s << scientific	采用科学记数法格式 d. ddddEdd
s << endl	输出 '\n'并清除缓冲
s << ends	输出 '\0'
s << flush	清除流
s >> ws	吃掉空白符
s << resetiosflags(f)	清空标志 f
s << setiosflags(f)	将标志设置为 f
s << setbase(b)	以基底 b 输出整数
s << setfill(c)	将填充字符设置为 c
s << setprecision(n)	精度设置为 n 个数字
s << setw(n)	下个域的宽度为 n 个字符

每个操作都返回指向第一个运算对象 s(流) 的引用，例如：

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

输出结果为：

```
1234,4d2,2322
```

而

```
cout << '(' << setw(4) << setfill('#') << 12 << ") (" << 12 << ")\n";
```

输出

```
(##12) (12)
```

为了显式设置浮点数的一般输出格式，我们可以使用

```
b.setf(ios_base::fmtflags(0), ios_base::floatfield)
```

请参见第 11 章。

B.8　字符串处理

标准库在 < cctype > 中提供了字符分类操作，在 < string > 中提供了字符串及相关操作，在 < regex > 中提供了正则表达式操作(C++0x)，在 < cstring > 中提供了 C 风格字符串的支持。

B.8.1　字符分类

基本字符集中的字符可分类如下：

字符分类	
isspace(c)	c 是空白符(' '、'\t'、'\n'等)吗
isalpha(c)	c 是字母('a'..'z'、'A'..'Z')吗(注意：不包括'_')
isdigit(c)	c 是十进制数字('0'..'9')吗
isxdigit(c)	c 是十六进制数字(十进制数字或 'a'..'f'或 'A'..'F')吗
isupper(c)	c 是大写字母吗
islower(c)	c 是小写字母吗
isalnum(c)	c 是字母或十进制数字吗
iscntrl(c)	c 是控制字符(ASCII 0..31 和 127)吗
ispunct(c)	c 不是字母、数字、空白符或可见控制字符吗
isprint(c)	c 是可打印的(ASCII' '..'~')吗
isgraph(c)	c 满足 isalpha()|isdigit()|ispunct()吗(注意：不包括空格)

另外，标准库提供了两个有用的函数，可以消除大小写差别：

大小写	
toupper(c)	返回 c 或者 c 的对应大写形式
tolower(c)	返回 c 或者 c 的对应小写形式

标准库还支持扩展字符集，如 Unicode，但这些内容已经超出了本书的范围。

B.8.2　字符串

标准库字符串类 string，是字符串模板 basic_string 对字符类型 char 的一个特例化，即 string 是 char 序列：

字符串运算	
s = s2	将 s2 赋予 s，s2 可以是字符串或者 C 风格字符串
s += x	将 x 附加到 s 的末尾，x 可以是字符、字符串或者 C 风格字符串
s[i]	下标

（续）

字符串运算	
s + s2	连接，结果字符串中的字符来自 s 后接来自 s2 的字符
s == s2	字符串比较，s 或 s2 可以为 C 风格字符串，但不能两者均是
s! = s2	字符串比较，s 或 s2 可以为 C 风格字符串，但不能两者均是
s < s2	字符串字典序比较，s 或 s2 可以为 C 风格字符串，但不能两者均是
s < = s2	字符串字典序比较，s 或 s2 可以为 C 风格字符串，但不能两者均是
s > s2	字符串字典序比较，s 或 s2 可以为 C 风格字符串，但不能两者均是
s > = s2	字符串字典序比较，s 或 s2 可以为 C 风格字符串，但不能两者均是
s. size( )	s 中字符数
s. length( )	s 中字符数
s. c_str( )	s 的 C 风格字符串版本(以 0 结束)
s. begin( )	指向首字符的迭代器
s. end( )	指向尾字符之后位置的迭代器
s. insert(pos, x)	将 x 插入到 s[pos]之前，x 可以是字符、字符串或 C 风格字符串
s. append(pos, x)	将 x 插入到 s[pos]之后，x 可以是字符、字符串或 C 风格字符串
s. erase(pos)	删除 s[pos]处的字符
s. push_back(c)	追加字符 c
pos = s. find(x)	在 s 中查找 x，x 可以是字符、字符串或 C 风格字符串。pos 为找到的第一个字符的下标，或者是 npos(s 末尾之后的位置)
in >> s	从 in 中读取一个词，存入 s

B. 8. 3　正则表达式匹配

正则表达式库现在还不是标准库的一部分，但很快就会被纳入标准库，而且其应用已经非常广泛，因此我们在此将其列出。更多细节请参考第 23 章。< regex > 的主要功能包括

- 在(任意长度的)数据流中搜索与正则表达式相匹配的字符串——regex_search( )
- 判定字符串(已知长度的)是否与正则表达式匹配——regex_match( )
- 替换匹配成功的串——regex_replace( )：本书中并未讨论，请参考专家级书籍或手册

regex_search( )或 regex_match( )的结果是一个匹配结果的集合，通常表示为 smatch：

```
regex row( "^[\\w ]+(          \\d+)( \\d+)( \\d+)$");      // data line

while (getline(in,line)) {      // check data line
        smatch matches;
        if (!regex_match(line, matches, row))
                error("bad line", lineno);
                                .
                // check row:
        int field1 = from_string<int>(matches[1]);
        int field2 = from_string<int>(matches[2]);
        int field3 = from_string<int>(matches[3]);
        // . . .
}
```

正则表达式的语法基于具有特殊含义的字符(参见第 23 章)：

正则表达式特殊字符	
.	任意单个字符(“通配符”)
[	字符分类
{	计数
(	分组开始
)	分组结束

（续）

正则表达式特殊字符	
\	转义字符——下一字符具有特殊含义
*	零次或多次重复
+	一次或多次重复
?	可选(零次或一次)
|	二选一(或)
^	行开始；非
$	行结束

重复	
{n}	重复 n 次
{n, }	重复 n 次或更多次
{n, m}	重复至少 n 次，至多 m 次
*	重复零次或多次，即{0, }
+	重复一次或多次，即{1, }
?	可选(零次或一次)，即{0, 1}

字符集	
alnum	任意字母数字字符或下划线
alpha	任意字母字符
blank	任意空白符，行分隔符除外
cntrl	任意控制字符
d	任意十进制数字
digit	任意十进制数字
graph	任意图形字符
lower	任意小写字符
print	任意可打印字符
punct	任意标点符号
s	任意空白符
space	任意空白符
upper	任意大写字符
w	任意单词字符(字母数字字符)
xdigit	任意十六进制数字字符

一些字符分类支持简写形式：

字符分类简写		
\d	十进制数字	[[: digit:]]
\l	小写字符	[[: lower:]]
\s	空白符(空格、制表符等)	[[: space:]]
\u	大写字符	[[: upper:]]
\w	字母、十进制数字或下划线(_)	[[: alnum:]]
\D	除\d 之外的任意字符	[^[: digit:]]
\L	除\l 之外的任意字符	[^[: lower:]]
\S	除\s 之外的任意字符	[^[: space:]]
\U	除\u 之外的任意字符	[^[: upper:]]
\W	除\w 之外的任意字符	[^[: alnum:]]

B.9　数值

C++对数学计算(如科学计算、工程计算等)提供最基础的支持。

B.9.1　数值限制

每个 C++ 编译器实现都指定了内置类型的属性，程序员可以使用这些属性来检查数值限制，设置“哨兵”(边界检测)等。

我们可以从 < limits > 中获得每个内置类型或者库类型 T 的限制 numeric_limits < T > 。此外，程序员还可以为用户自定义类型 X 定义限制 numeric_limits < X > 。例如：

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2;     // base of exponent (in this case, binary)
    static const int digits = 24;   // number of radix digits in mantissa
    static const int digits10 = 6; // number of base-10 digits in mantissa

    static const bool is_signed = true;
    static const bool is_integer = false;
    static const bool is_exact = false;

    static float min() { return 1.17549435E-38F; }     // example value
    static float max() { return 3.40282347E+38F; }     // example value

    static float epsilon() { return 1.19209290E-07F; }    // example value
    static float round_error() { return 0.5F; }           // example value

    static float infinity() { return /* some value */; }
    static float quiet_NaN() { return /* some value */; }
    static float signaling_NaN() { return /* some value */; }
    static float denorm_min() { return min(); }

    static const int min_exponent = -125;          // example value
    static const int min_exponent10 = -37;         // example value
    static const int max_exponent = +128;          // example value
    static const int max_exponent10 = +38;         // example value

    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;

    static const bool is_iec559 = true;       // conforms to IEC-559
    static const bool is_bounded = true;
    static const bool is_modulo = false;
    static const bool traps = true;
    static const bool tinyness_before = true;

    static const float_round_style round_style = round_to_nearest;
};
```

从 < limits. h > 和 < float. h > 中，我们可以获得指明整数和浮点数的一些关键属性的宏，包括：

数值限制宏	
CHAR_BIT	一个 char 中的二进制位数目(通常为 8)
CHAR_MIN	char 的最小值
CHAR_MAX	char 的最大值(如果 char 是有符号的，通常为 127；如果 char 是无符号的，通常为 255)
INT_MIN	int 的最小值

（续）

数值限制宏	
INT_MAX	int 的最大值
LONG_MIN	int 的最小值
LONG_MAX	int 的最大值
FLT_MIN	float 的最小正值（如 1.175494351e－38F）
FLT_MAX	float 的最大值（如 3.402823466e＋38F）
FLT_DIG	精度为多少个十进制数字（如 6）
FLT_MAX_10_EXP	最大的十进制指数（如 38）
DBL_MIN	double 的最小值
DBL_MAX	double 的最大值（如 1.7976931348623158e＋308）
DBL_EPSILON	使 1.0＋DBL_EPSILON ！＝ 1.0 成立的最小值

B.9.2　标准数学函数

标准库提供了最常用的数学函数（定义于 <cmath> 和 <complex>）：

标准数学函数	
abs(x)	绝对值
ceil(x)	>= x 的最小整数
floor(x)	<= x 的最大整数
sqrt(x)	平方根，x 必须是非负数
cos(x)	余弦
sin(x)	正弦
tan(x)	正切
acos(x)	反余弦，结果是非负数
asin(x)	反正弦，返回最接近 0 的值
atan(x)	反正切
sinh(x)	双曲正弦
cosh(x)	双曲余弦
tanh(x)	双曲正切
exp(x)	基底为 e 的指数
log(x)	自然对数，基底为 e，x 必须是正数
log10(x)	基底为 10 的对数

这些函数都有分别针对 float、double、long double 和 complex 的不同版本。对每个函数，返回值类型与参数类型一致。

如果一个标准数学函数不能生成数学上有效的结果，它会恰当设置 errno。

B.9.3　复数

标准库提供了复数类型 complex<float>、complex<double> 和 complex<long double>。如果 Scalar 是其他某种支持常用算术运算的类型，complex<Scalar> 通常也能正常工作，但不保证是可移植的。

```
template<class Scalar> class complex {
    // a complex is a pair of scalar values, basically a coordinate pair
    Scalar re, im;
public:
    complex(const Scalar & r, const Scalar & i) :re(r), im(i) { }
    complex(const Scalar & r) :re(r),im(Scalar ()) { }
    complex() :re(Scalar ()), im(Scalar ()) { }

    Scalar real() { return re; }      // real part
    Scalar imag() { return im; }      // imaginary part

    // operators: = += -= *= /=
};
```

除了复数的成员之外，<complex> 还提供了大量有用的操作：

复数运算符	
z1 + z2	加
z1 - z2	减
z1 * z2	乘
z1/z2	除
z1 == z2	相等
z1 ! = z2	不相等
norm(z)	abs(z)的平方
conj(z)	共轭：若 z 为{re, im}，则 conj(z)为{re, -im}
polar(x, y)	由极坐标(rho, theta)构造一个复数
real(z)	复数的实部
imag(z)	虚部
abs(z)	也称为 rho
arg(z)	也称为 theta
out << z	输出复数
in >> z	输入复数

标准数学函数(参见附录 B.9.2)也都有复数版本。注意，complex 不提供 < 或% ，参见 24.9 节。

B.9.4　valarray

标准库 valarray 是一维数值数组，即它为数组类型提供了算术运算，并提供了对子数组和跨越访问的支持。

B.9.5　泛型数值算法

下面函数均来自 < numeric > ，它们提供了数值序列的通用运算的泛型版本：

数值算法	
x = accumulate(b, e, i)	x 是 i 和[b:e)中元素之和
x = accumulate(b, e, i, f)	累计，但使用 f 代替 +
x = inner_product(b, e, b2, i)	x 是[b:e)和[b2:b2 + (e - b))的内积，即 x 是 i 和所有(*p1) * (*p2)之和，p1 是[b:e)中的元素，p2 是[b2:b2 + (e - b))中对应的元素
x = inner_product(b, e, b2, i, f, f2)	内积，但分别用 f 和 f2 代替 + 和*
p = partial_sum(b, e, out)	[out:p)中第 i 个元素是[b:e)中第 0 个到第 i 个元素之和
p = partial_sum(b, e, out, f)	部分和，使用 f 代替 +
p = adjacent_difference(b, e, out)	对 i > 0，[out:p)中第 i 个元素是*(b + i) - *(b + i - 1)；若 e - b > 0，则*out 是*b
p = adjacent_difference(b, e, out, f)	相邻差，用 f 代替 -

B.10　C 标准库函数

C 语言标准库在集成入 C++ 标准库时只进行了很小的修改。C 标准库中提供的函数，都是在大量不同实际环境中，特别是相对低层的程序设计领域，经过长期验证，被证明是非常有用的函数。在本节中，我们将它们组织为几个传统的类别来进行介绍：

- C 风格 I/O
- C 风格字符串
- 内存
- 日期和时间
- 其他

还有很多 C 标准库函数本节没有介绍，如果你需要了解这些函数，请参考一本好的教材，如 Ker-

nighan 和 Ritchie 的《The C++ Programming Language》(K&R)。

B.10.1　文件

<stdio>定义的 I/O 系统是基于“文件”的。文件(FILE*)可以指向文件或者一个标准输入/输出流 stdin、stdout 和 stderr。标准流可以直接使用,其他文件需要显式打开:

文件打开和关闭	
f = fopen(s, m)	以模式 m 打开一个文件 s
x = fclose(f)	关闭文件 f,如果成功则返回 0

“模式”是一个字符串,包含一个或多个指明文件如何打开的指令:

文件模式	
"r"	读
"w"	写(丢弃已有内容)
"a"	追加(添加到末尾)
"r+"	读和写
"w+"	读和写(丢弃已有内容)
"b"	二进制模式,与其他一个或多个模式一起使用

在一个特定系统中,可能有更多模式(通常也确实是)。一些模式可以组合,例如,fopen("foo", "rb")试图打开文件 foo 来进行二进制读。stdio 和 iostream(参见附录 B.7.1)的 I/O 模式应该是相同的。

B.10.2　printf()函数家族

最常用的 C 标准库函数是 I/O 函数。但是,我们推荐使用 iostream,因为它是类型安全且可扩展的。格式化输出函数 printf()应用非常广泛(在 C++程序中使用也非常广泛),也被其他程序设计语言广泛模仿。

printf	
n = printf(fmt, args)	将参数 args 恰当地插入“格式串”fmt 中,将其输出到 stdout
n = fprintf(f, fmt, args)	将参数 args 恰当地插入“格式串”fmt 中,将其输出到文件 f
n = sprintf(s, fmt, args)	将参数 args 恰当地插入“格式串”fmt 中,将其输出到 C 风格字符串 s

对于每个版本,n 返回打印的字符数,或者是一个负数表示输出错误。printf()的返回值一般应该忽略。

printf()的声明是这样的

```
int printf(const char* format ...);
```

换句话说,它接受一个 C 风格字符串(通常是一个字符串文字常量),后跟任意数量、任意类型的参数。这些“额外参数”的含义由格式串中的转换说明符,如% c(打印字符)和% d(打印十进制整数)来控制。例如:

```
int x = 5;
const char* p = "asdf";
printf("the value of x is '%d' and the value of p is '%s'\n",x,s);
```

% 后跟一个字符控制了如何处理参数。第一个% 应用于第一个“额外参数”(本例中,% d 应用于 x),第二个% 应用于第二个“额外参数”(本例中,% s 应用于 s),依此类推。本例中 printf()的输出结果为

```
the value of x is '5' and the value of p is 'asdf'
```

其后接一个换行。

一般来说，无法检查% 转换指令与其所应用的参数类型之间的对应关系，即便可以，printf( ) 通常也不会检查。例如：

```
printf("the value of x is '%s' and the value of p is '%d'\n",x,p);  // oops
```

C 标准库提供了大量转换说明符，从而提供了极大的灵活性(另一方面也容易导致混淆)：

–	可选的减号，指明输出值在域中左对齐
+	可选的加号，指明有符号类型的值总是以 + 或 – 开始，来表明正负值
0	可选的 0，指明使用前导 0 来对数值进行填充。如果指定了 – 或精度，则忽略 0
#	可选的#，指明浮点值必须打印小数点，即便之后没有非 0 数字——这种情况打印结尾 0，它的另外两种含义是八进制数打印一个前缀 0，以及十六进制数打印前缀 0x 或 0X
d	可选的数字串，指明域宽，如果输出值的字符数小于域宽，则在左侧填充空格(或者当指定了左对齐指示符时，在右侧填充空格)；如果域以 0 开始，则填充 0 而不是空格
.	可选的句点，用来将域宽与下一个数字串分开
dd	可选的数字串，指明精度：对于转换符 e 和 f 来说，指出小数点后有多少位数字；或者是一个字符串中最多打印多少个字符
*	域宽和精度可以用 * 而不是数字串来指定，在这种情况下，额外参数中的一个整型值用来指定域宽和精度
h	可选的字母 h，指明后面的 d、o、x 或 u 对应短整型参数
l	可选的字母 l，指明后面的 d、o、x 或 u 对应长整型参数
L	可选的字母 L，指明后面的 e、E、g、G 或 f 对应 long double 型参数
%	指出打印一个字符% ，不使用任何参数
c	一个指明转换类型的字符，C 标准库提供的转换符及其含义如下：
d	将整数参数转换为十进制表示
i	将整数参数转换为十进制表示
o	将整数参数转换为八进制表示
x	将整数参数转换为十六进制表示
X	将整数参数转换为十六进制表示
f	将 float 或 double 参数转换为十进制表示，格式为[–]*ddd. ddd*。小数点之后的数字个数与参数对应的精度相等。如果需要的话，对数值进行舍入。如果未指定精度，则打印小数点后六位。如果显式地指定了精度为 0，且未指定#，则不打印小数点
e	将 float 或 double 参数转换为十进制表示，格式为科学计数法[–]*d. ddde + dd* 或者[–]*d. ddde – dd*，小数点之前只有一位数字，小数点之后的数字个数与参数对应的精度相等。如果需要的话，对数值进行舍入。如果未指定精度，则打印小数点后六位。如果显式地指定了精度为 0，且未指定#，则不打印数字和小数点。
E	与 e 相同，但指数之前用大写的 E

g	将 float 或 double 参数打印为格式 d、f 或 e，哪种格式能以最少的空间表示最大的精度，就采用哪种格式
G	与 g 相同，但指数之前用大写的 E
c	打印一个字符参数，忽略空字符
s	接受一个字符串参数(字符指针)，打印其中的字符，直至遇到空字符或者达到精度上限。但是，如果精度为 0 或未指定精度，则空字符之前的所有字符均被打印
p	打印一个指针，打印格式依赖于具体实现
u	将无符号整数转换为十进制表示
n	将 printf( )、fprintf( )或 sprintf( )所打印的字符数写入整型指针参数指向的地址

如果未指定域宽，或者指定的域宽很小，不会对域进行截断；只有当指定域宽超出实际域宽时，才会进行填充。

由于 C 的用户自定义类型与 C++ 含义不同，因此不能为 complex、vector 或 string 等用户自定义类型定义输出格式。

C 的标准输出 stdout 与 cout 相对应。C 的标准输入 stdin 与 cin 相对应。C 的标准错误输出 stderr 与 cerr 相对应。C 标准 I/O 和 C++ I/O 流之间的关系是非常紧密的，甚至两者的缓冲区都是共享的。例如，混合使用 cout 和 stdout 可以生成单一的输出流(这在 C/C++ 混合程序中并不罕见)。但这种灵活性也会带来额外的代价，出于性能考虑，不要对同一个流混合使用 stdio 和 iostream 操作，并且在第一个 I/O 操作前调用 ios_base :: sync_with_stdio(false)。

stdio 库提供了一个名为 scanf( )的函数，它是输入操作，风格与 printf( )类似。例如：

```
int x;
char s[buf_size];
int i = scanf("the value of x is '%d' and the value of s is '%s'\n",&x,s);
```

在这段代码中，scanf( )试图读取一个整数存入 x，并读取一个非空白符字符序列存入 s。格式串中包含一些非格式控制字符，用户的输入中必须包含这些字符，如输入以下内容：

```
the value of x is '123' and the value of s is 'string '\n"
```

则上面例程会读取 123 存入 x，并将 string 后跟一个 0 存入 s。如果 scanf( )操作成功，返回结果(上例中的 i)将是成功赋值的参数指针数(上例中期望是 2)；否则，返回 EOF 表示失败。这种指定输入格式的方式很容易出错(例如，在上例中，如果忘记输入 string 后面的空格，会发生什么情况)。传递给 scanf( )的参数必须都是指针，我们强烈建议不使用 scanf( )。

那么，如果不得不使用 stdio，如何来进行输入呢？一个常见的回答是“使用标准库函数 gets( )”：

```
// very dangerous code:
char s[buf_size];
char* p = gets(s);      // read a line into s
```

p = gets(s)会一直读取字符存入 s，直至遇到换行或者文件尾，在最后一个字符之后放置一个 0。如果开始就遇到文件尾，或者发生了错误，则将 p 置为 NULL(即 0)；否则将 s 赋予 p。不要使用 gets(s)或大致等价的 scanf("%s", s)！长久以来，它们一直是病毒设计者的最爱：通过输入大量数据，使缓冲区(上例中的 s)溢出，黑客就可以使程序崩溃甚至接管计算机。函数 sprintf( )也存在这种缓冲区溢出问题。

stdio 库也提供了简单有效的字符输入/输出函数：

stdio 字符函数	
x = getc(st)	从输入流 st 读入一个字符，返回字符的整型值；如果到达文件尾或发生错误返回 EOF
x = putc(c, st)	将字符 c 写入输出流 st，返回 c 的整型值；若发生错误返回 EOF
x = getchar(c)	从 stdin 读取一个字符，返回字符的整型值，遇到文件尾或发生错误返回 EOF
x = putchar(c)	将字符 c 写入 stdout，返回字符的整型值，若发生错误返回 EOF
x = ungetc(s, t)	将 c 退回输入流 st，返回字符的整型值，若发生错误返回 EOF

注意，这些函数的返回值是 int(而不是 char，否则就不能返回 EOF 了)。例如，下面是一个典型的 C 风格输入循环：

```
int ch;  /* not char ch; */
while ((ch=getchar())!=EOF) { /* do something */ }
```

不要对一个流连续做两次 ungetc()，这种操作方式的结果是未定义的，因此这种代码是不具有可移植性的。

除上述函数之外，stdio 库还包含其他很多函数，如果你希望了解更多，请参考 K&R 这类好的 C 教材。

B.10.3　C 风格字符串

C 风格字符串是以 0 结尾的字符数组。支持这种字符串表示法的函数都定义于 < cstring > (或 < string.h > ，注意不是 < string >)和 < cstdlib > 中。这些函数对 char* 指针指向的字符串进行操作(如果是 const char* ，则是只读的)：

C 风格字符串操作	
x = strlen(s)	统计字符数(不包括结尾 0)
p = strcpy(s, s2)	将 s2 拷贝到 s；[s: s + n)和[s2: s2 + n)两个区间不能重叠；将 s 返回给 p；结尾 0 也拷贝
p = strcat(s, s2)	将 s2 拷贝到 s 末尾；将 s 返回给 p；结尾 0 也拷贝
x = strcmp(s, s2)	字典序比较：如果 s < s2 返回负值；若 s == s2 返回 0；若 s > s2 返回正值
p = strncpy(s, s2, n)	类似 strcpy；最多拷贝 n 个字符；可能无法拷贝结尾 0；返回 s
p = strncat(s, s2, n)	类似 strcat；最多连接 n 个字符；可能无法拷贝结尾 0；返回 s
x = strncmp(s, s2, n)	类似 strcmp；最多比较 n 个字符
p = strchr(s, c)	返回指针，指向 s 中 c 第一次出现的位置
p = strrchr(s, c)	返回指针，指向 s 中 c 最后一次出现的位置
p = strstr(s, s2)	返回指针，指向 s 中与 s2 相等的字串的首字符
p = strpbrk(s, s2)	返回指针，指向 s 中第一个也在 s2 中出现的字符
x = atof(s)	从 s 中提取出一个 double 值
x = atoi(s)	从 s 中提取出一个 int 值
x = atol(s)	从 s 中提取出一个 long int 值
x = strtod(s, p)	从 s 中提取出一个 double 值，p 指向 double 值之后的第一个字符
x = strtol(s, p)	从 s 中提取出一个 long int 值，p 指向 long int 值之后的第一个字符
x = strtoul(s, p)	从 s 中提取出一个 unsigned long int 值，p 指向 unsigned long int 值之后的第一个字符

注意，在 C++ 中，strchr()和 strstr()都有两个版本来实现类型安全(它们不能将 const char* 转换为 char* ，而 C 版本是可以的)，参见 27.5 节。

最后几个函数在 C 风格字符串内查找常规的数值表示形式，如“124”和“　1.4”。如果未找到，则返回 0。例如：

```
int x = atoi("fortytwo");     /* x becomes 0 */
```

B.10.4　内存

内存处理函数通过 void* 指针在“原始内存”(类型未知的内存区域)上进行操作(const void* 指针对应只读内存):

C 风格内存操作	
q = memcpy(p, p2, n)	从 p2 开始拷贝 n 个字节到 p(类似 strcpy);[p: p + n)和[p2: p2 + n)两个区间不能重叠;将 p 返回给 q
q = memmove(p, p2, n)	从 p2 开始拷贝 n 个字节到 p;将 p 返回给 q
x = memcmp(p, p2, n)	比较 p2 开始的 n 个字节和 p 中对应的 n 个字节(类似 strcmp)
q = memchr(p, c, n)	在 p[0]..p[n-1]中查找 c(转换为 unsigned char),若找到,返回指向该字节的指针,否则返回 0
q = memset(p, c, n)	将 c(转换为 unsigned char)赋予 p[0]..p[n-1]中每个字节,返回 p
p = calloc(n, s)	在自由内存空间中分配 n* s 个字节,全部初始化为 0;若分配失败返回 0
p = malloc(s)	在自由内存空间中分配 s 个字节,不进行初始化,若分配失败返回 0
q = realloc(p, s)	在自由内存空间中分配 s 个字节;p 必须是 malloc()或 calloc()返回的指针;如果可能,继续使用 p 指向的空间。如果不行,将 p 指向的内存空间中的所有字节都拷贝到新空间中;若分配失败返回 0
free(p)	释放 p 指向的内存空间;p 必须是 malloc()、calloc()或 realloc()返回的指针

注意,malloc()等函数不调用构造函数,free 也不调用析构函数。对具有构造函数和析构函数的类型,不要使用这些函数。同样,对具有构造函数的类型也不要使用 memset()。

以 mem* 开头的函数都定义于 < cstring > ,而分配函数都在 < cstdlib > 中定义。

同样参见 27.5.2 节。

B.10.5　日期和时间

< ctime > 中定义了一些日期和时间相关的类型和函数。

日期和时间类型	
clock_t	算术类型,用于保存较短的时间间隔(可能只是几分钟的间隔)
time_t	算术类型,用于保存较长的时间间隔(可能几个世纪)
tm	结构类型,保存(自 1900 年以来的)日期和时间

struct tm 定义如下:

```
struct tm {
    int tm_sec;   // second of minute [0:61]; 60 and 61 represent leap seconds
    int tm_min;   // minute of hour [0,59]
    int tm_hour;  // hour of day [0,23]
    int tm_mday;  // day of month [1,31]
    int tm_mon;   // month of year [0,11]; 0 means January (note: not [1:12])
    int tm_year;  // year since 1900; 0 means year 1900, and 102 means 2002
    int tm_wday;  // days since Sunday [0,6]; 0 means Sunday
    int tm_yday;  // days since January 1 [0,365]; 0 means January 1
    int tm_isdst; // hours of Daylight Savings Time
};
```

日期和时间函数如下:

```
clock_t clock();      // number of clock ticks since the start of the program

time_t time(time_t* pt);     // current calendar time
double difftime(time_t t2, time_t t1);  // t2-t1 in seconds
```

```
tm* localtime(const time_t* pt);  // local time for the *pt
tm* gmtime(const time_t* pt); // Greenwich Mean Time (GMT) tm for *pt, or 0

time_t mktime(tm* ptm);          // time_t for *ptm, or time_t(-1)

char* asctime(const tm* ptm);    // C-style string representation for *ptm
char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

调用 asctime( )返回的结果可能是"Sun Sep 16 01：03：52 1973\n"这样的字符串。

下例说明了如何使用 clock 来对函数计时(do_something( ))：

```
int main(int argc, char* argv[])
{
    int n = atoi(argv[1]);

    clock_t t1 = clock();       // start time
    if (t1 == clock_t(-1)) {    // clock_t(-1) means "clock() didn't work"
        cerr << "sorry, no clock\n";
        exit(1);
    }
    for (int i = 0; i<n; i++) do_something();   // timing loop

    clock_t t2 = clock();       // end time
    if (t2 == clock_t(-1)) {
        cerr << "sorry, clock overflow\n";
        exit(2);
    }
    cout << "do_something() " << n << " times took "
        << double(t2-t1)/CLOCKS_PER_SEC << " seconds"
        << " (measurement granularity: " << CLOCKS_PER_SEC
        << " of a second)\n";
}
```

除法运算之前的显式类型转换 double(t2 - t1)是必须的，因为 clock_t 可能是一个整数。对于 clock ( )返回的 t1 和 t2，double(t2 - t1)/CLOCKS_PER_SEC 是系统能达到的两次调用间时间间隔的最佳近似。

如果处理器未提供 clock( )功能，或者时间间隔太长，clock( )会返回 clock_t(-1)。

B. 10. 6　其他函数

在 < cstdlib > 中定义了如下函数：

其他 stdlib 函数	
abort( )	"非正常"结束程序
exit(n)	结束程序，返回 n；n == 0 表示程序运行成功
system(s)	将 C 风格字符串参数作为命令执行(具体行为依赖于系统)
qsort(b, n, s, cmp)	使用比较函数 cmp 排序从 b 开始的 n 个元素，元素大小为 s
bsearch(k, b, n, s, cmp)	使用比较函数 cmp 在 b 开始的 n 个元素中搜索 k，元素大小为 s
d = rand( )	d 为范围[0：RAND_MAX]之间的伪随机数
srand(d)	用 d 作为种子，开始一个伪随机数序列

qsort( )和 bsearch( )所使用的比较函数(cmp)必须是下面类型：

```
int (*cmp)(const void* p, const void* q);
```

也就是说，排序函数并不了解类型信息，它将数组简单看做字节序列。cmp 返回值的含义为：

- 负数表示*p 小于*q
- 0 表示*p 等于*q

- 正数表示*p 大于*q

注意，exit( )和 abort( )不调用析构函数。如果你希望对构造的自动对象和静态对象调用析构函数(参见附录 A.4.2)，应抛出一个异常。

标准库的更多内容，请参考 K&R 或其他富有声誉的 C 语言参考资料。

B.11　其他库

当你浏览标准库功能，毫无疑问可能找不到一些你需要的功能。与程序员面临的挑战以及世界上已有的众多库相比，C++是很渺小的。还存在很多其他用途的库：

- 图形用户界面库
- 高级数学库
- 数据库访问库
- 网络库
- XML 库
- 日期和时间库
- 文件系统处理库
- 三维图形库
- 动画库
- 其他用途的库

但是，这些库不是标准库的一部分。你可以通过搜索互联网或者请教朋友和同事来寻找这些库。请不要形成这样一种观念：所有有用的库都是标准库的一部分。